TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Facial Expression Detection using Convolutional Neural Networks

TITULACIÓN: Grau en Enginyeria de Sistemes de Telecomunicació

AUTOR: Albert Vilagran Solsona

DIRECTOR: Francesc Tarrés

FECHA: 24 de octubre del 2018

Título: Facial Expression Detection using Convolutional Neural Networks

Autor: Albert Vilagran Solsona

Director: Francesc Tarrés

Fecha: 24 de octubre del 2018

Resumen

El mundo de la inteligencia artificial avanza a gran velocidad. El mayor ejemplo de ello son las redes neuronales, que están revolucionando el sistema de la automatización y de la extracción de datos.

Las redes neuronales pueden ser entrenadas para realizar diferentes tareas clasificatorias. Este proyecto se enfoca en el reconocimiento de 7 expresiones faciales aplicando arquitecturas de redes neuronales convolucionales.

El objetivo de este proyecto consistirá en una serie de pasos:

En primer lugar, el estudio teórico de las redes neuronales convolucionales y sus parámetros, de tal manera que se pueda optimizar al máximo la precisión del sistema realizando procesos de entrenamiento con una base de datos.

Una vez hayamos obtenido la mejor arquitectura para dicho propósito, se procederá a realizar varios test manuales sobre ese sistema para verificar su funcionamiento en función de la precisión que nos ofrezca.

Finalmente, gracias a una base de datos proporcionada, realizaremos un estudio de las expresiones faciales de personajes del paradigma político, de tal manera que pueda proyectarse a una aplicación real.

Title: Facial Expression Detection using Convolutional Neural Networks

Author: Albert Vilagran Solsona

Director: Francesc Tarrés

Date: October 24 th 2018

Overview

The world of artificial intelligence advances at great speed. The greatest example of this is neural networks, which are revolutionizing the system of automation and data extraction.

Neural networks can be trained to perform different classifying tasks. This project focuses on the recognition of 7 facial expressions by applying architectures of convolutional neural networks.

The objective of this project will consist of a series of steps:

First, the theoretical study of convolutional neural networks and their parameters, in such a way that the accuracy of the system can be optimized by carrying out training processes with a database.

Once we have obtained the best architecture for that purpose, we will proceed to perform several manual tests on that system to verify its operation according to the accuracy it offers us.

Finally, thanks to a database provided, we will carry out a study of the facial expressions of characters of the political paradigm, in such a way that it can be projected to a real application.

ÍNDICE

Contenido

INTRODUCCIÓN

Es bien conocido que la comunicación no verbal de una persona ronda más o menos alrededor del 55%, la cual incluye señales, gestos y las expresiones faciales. Es decir, los humanos somos capaces de extraer más de la mitad de la información de una comunicación a través del sentido de la visión. ¿Podríamos ser capaces de dotar a una maquina la capacidad de obtener, procesar y extraer esta información para nuestros propósitos?

Los análisis de expresión facial mediante el uso de redes neuronales son objeto de estudio a lo largo de distintos artículos y concursos que se han realizado para obtener la mayor precisión a la hora de obtener esta información.

Los aplicativos de obtener una red de estas son muy diversos, pero el principal que podríamos nombrar es el *neuromarketing*, donde los análisis de estos datos son de vital importancia para analizar los patrones del consumidor de manera automática.

El objetivo de este proyecto se enfoca en la realización de una arquitectura de una red neuronal convolucional para dicho propósito. El proyecto está estructurado en cuatro capítulos.

En el primer capítulo, se da una visión teórica sobre los elementos que conforman una red neuronal convolucional, partiendo desde la unidad más pequeña, la neurona artificial, luego en como esta neurona puede ‘aprender’ de distintas formas con las funciones de activación, hasta la conformación de capas y arquitecturas disponibles para el proyecto.

En el segundo capítulo, se realiza un estudio teórico sobre los más importantes parámetros de configuración de la red neuronal que hayamos construido. Los algoritmos de optimización son descritos en detalle para la correcta elección posterior.

En el tercer capítulo, se construye la red neuronal y se muestran los principales resultados obtenidos con cada arquitectura. La arquitectura que haya resultado mejor para el propósito será posteriormente puesta a prueba por imágenes propias.

Finalmente, en el cuarto capítulo, superando los objetivos, le daremos una aplicación real a nuestra red neuronal y se expondrán los resultados que nos proporciona.

El proyecto está basado en el lenguaje de programación Python haciendo uso de las librerías Keras que implementan Tensorflow, código desarrollado por Google. Se utiliza el entorno de desarrollo integrado PyCharm que ejecuta nuestro código. Se implementa también la librería OpenCV para la detección de caras.

Capítulo 1. Elementos de una red neuronal

1.1. Introducción

Hay diversos tipos de redes neuronales, pero en este apartado nos enfocaremos en describir los elementos que forman parte de una red neuronal convolucional (CNN), sus diversas etapas y las arquitecturas propuestas para este proyecto.

1.2. Elementos de una red neuronal convolucional

1.2.1. La neurona

La neurona artificial es la unidad mínima dentro de una red neuronal artificial. Normalmente se hace el símil a una neurona biológica para explicar sus partes.

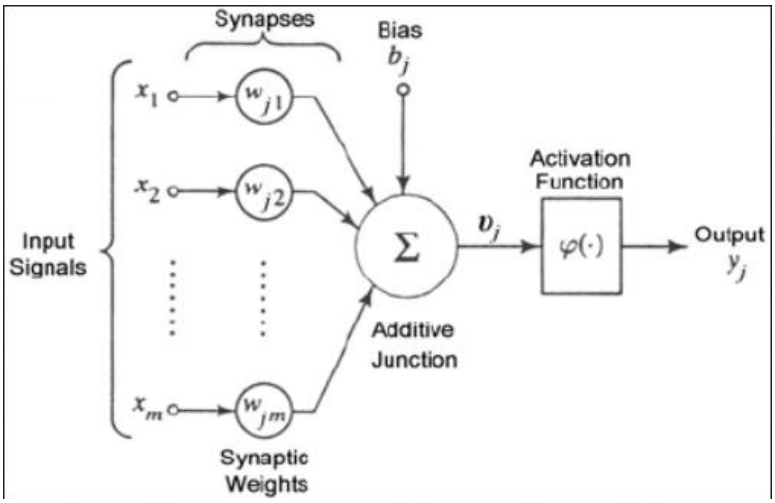


Fig. 1.1. Esquema y partes de una neurona artificial.

Esta se compone por los valores de entrada $X_{m,}$ que son recibidos por las conexiones sinápticas. En cada una de estas conexiones sinápticas hay asociado un peso W_{jm}. Este peso es el factor de importancia, que es modificado a lo largo del entrenamiento de la red neuronal. Entonces todos estos términos se les realiza una suma ponderada **(1.1)**.

$$h_i(t) = \sum_j W_{jm} X_m \qquad \textbf{(1.1)}$$

A continuación, cuando esta operación sea realizada, solo si se supera un umbral B_j, esta neurona emitirá dicho resultado a través de su salida Y_j. Finalmente hay la etapa de la función de activación $\varphi\,(\cdot)$, que es una función no lineal que funciona como umbral para realizar redes neuronales no lineales. En el siguiente apartado se explicará las funciones de activación que hay.

1.2.2. Funciones de activación

1.2.2.1. Función Sigmoide

Esta función es un caso especial de la función logística **(1.2)**. Dónde *L* es el máximo de la función y *K* es lo abrupta que es la curva. Definiendo esta *L = 1* y *K = 1* obtendríamos la función sigmoide **(1.3)**.

$$\sigma(\mathrm{x}) = \frac{L}{1+e^{-k(x-x_0)}} \qquad \textbf{(1.2)}$$

$$\sigma(\mathrm{x}) = \frac{1}{1+e^{-x}} \qquad \textbf{(1.3)}$$

Los números grandes negativos se convierten en 0 y los números grandes positivos en 1. Además, la función sigmoide tiene una buena interpretación como la velocidad de activación de una neurona. Pasa desde no activarse en absoluto (0), hasta el activarse completamente (1). Sin embargo, tiene dos grandes inconvenientes.

En primer lugar, la sigmoide satura y acaba con los gradientes. Cuando la neurona satura en 0 o 1, gradiente en estas zonas es prácticamente cero. En el proceso de *Backpropagation*, proceso en el cual el error de una capa de neuronas se propaga a la anterior capa para realizar una corrección de los pesos de esta misma. Entonces si este gradiente local es muy pequeño, acabará con el gradiente general y no habrá señal a través de la neurona y sus pesos. Entonces es de vital importancia inicializar los pesos de las neuronas sigmoideas para evitar la saturación. Pero esos pesos no deben ser demasiado grandes, sino la mayoría de neuronas se saturarán y la red no aprenderá.

El segundo problema se trata de que las salidas de las neuronas sigmoideas no están centradas a cero y eso no es lo deseable ya que las neuronas en las capas posteriores deberían recibir estas entradas centradas a cero. Esto puede resultar en un movimiento zigzag del gradiente para los pesos.

1.2.2.2. Función tangente hiperbólica

La función tangente hiperbólica es una alternativa a la función sigmoide, su fórmula **(1.3)** se define de la siguiente manera.

$$\tanh(\mathrm{x}) = \frac{1-e^{-2x}}{1+e^{-2x}} \qquad \textbf{(1.3)}$$

Como se puede apreciar la gráficas comparación **(1.2)** y **(1.3)** función tangente hiperbólica es muy parecida, pero con una diferencia que arregla uno de los problemas anteriores. Las salidas de las neuronas están centradas a cero. Con lo cual es preferible el uso de este tipo de neuronas en la práctica.

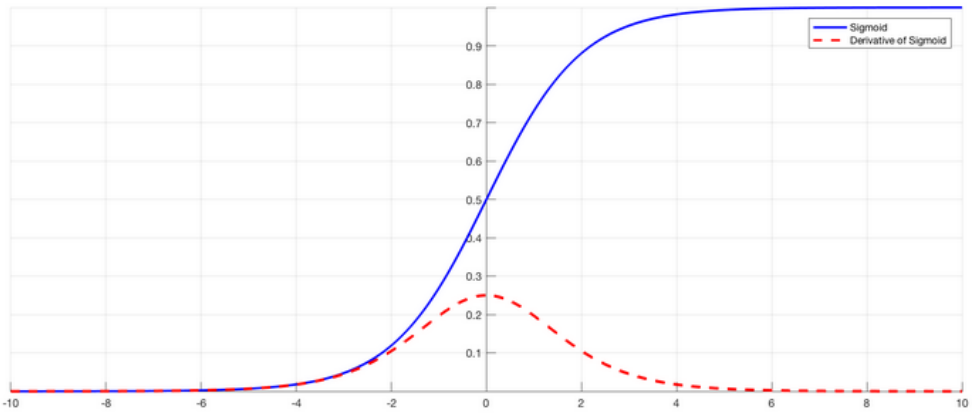


Fig. 2.2. Gráfico de la función sigmoide.

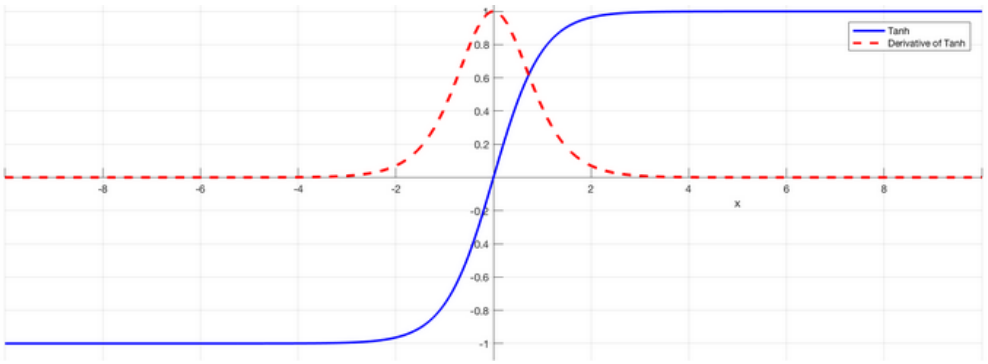


Fig. 3.3. Gráfico de la función tangente hiperbólica.

1.2.2.3. Unidad lineal Rectificada (ReLU)

La función de activación ReLU está definida por la siguiente expresión **(1.4)**:

$$f(x) = \max(0, x) \qquad \textbf{(1.4)}$$

Donde X es la entrada de la neurona. Es decir, la activación es llevada a cabo cuando pasa de cero. Se puede apreciar esta afirmación en el siguiente gráfico **(1.4)**.

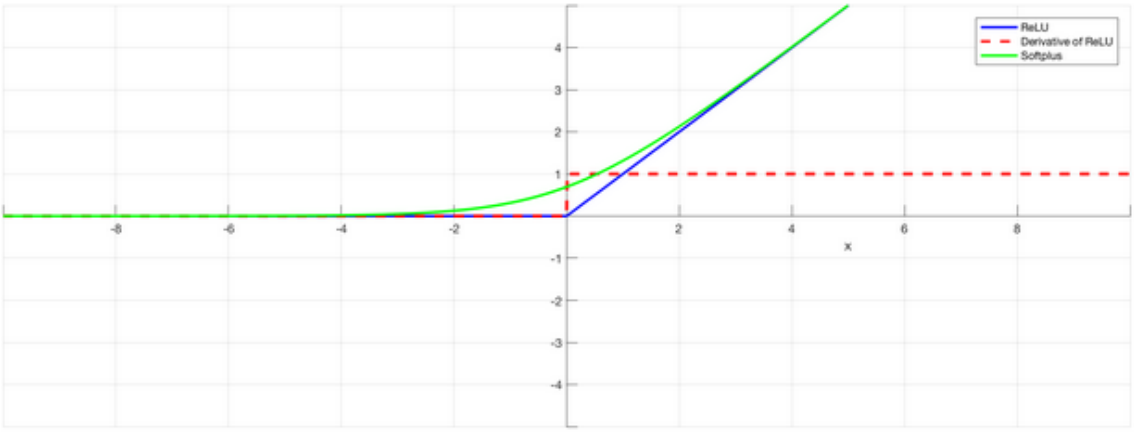


Fig. 4.4. Gráfico de la función ReLU.

La función ReLU es más efectiva que la sigmoide y la alternativa más práctica, la tangente hiperbólica, ya que reduce eficazmente el costo de cálculo. Sin embargo, las neuronas con función ReLU presentan varios inconvenientes. En primer lugar, las neuronas durante el entrenamiento pueden quedar inutilizadas debido a que los pesos puedan ser actualizados de tal manera que la neurona no vuelva a ser activada nuevamente. Para evitar este problema, hay que elegir una tasa de aprendizaje lo suficientemente pequeña.

1.2.2.4. Leaky/paramétrica ReLU

La Leaky ReLU es propuesta a la solución propuesta a la función de activación clásica ReLU. Su diferencia reside en que, mientras ReLU es cero para todo número más pequeño a cero *(X<0)*, Leaky ReLU añade una pendiente *α* para estos números por debajo de cero **(1.5)**. Este parámetro *α* es una constante más pequeña a 1. El caso concreto que esta pendiente es configurada para cada neurona, la función es llamada paramétrica[1].

$$\begin{cases} f(x) = \alpha x, & (x < 0) \\ f(x) = x, & (x \geq 0) \end{cases} \qquad \textbf{(1.5)}$$

1.2.2.5. ReLU randomizada

La ReLU randomizada es una vertiente de la Leaky ReLU. En este caso se elige una pendiente aleatoria dentro de un rango, determinado dentro de una función de distribución uniforme, dentro del proceso entrenamiento. En el proceso de testing, esta pendiente es fijada. A continuación, se muestran las gráficas de las distintas ReLU expuestas **(1.5)**.

[1] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Retrieved from http://arxiv.org/abs/1502.01852

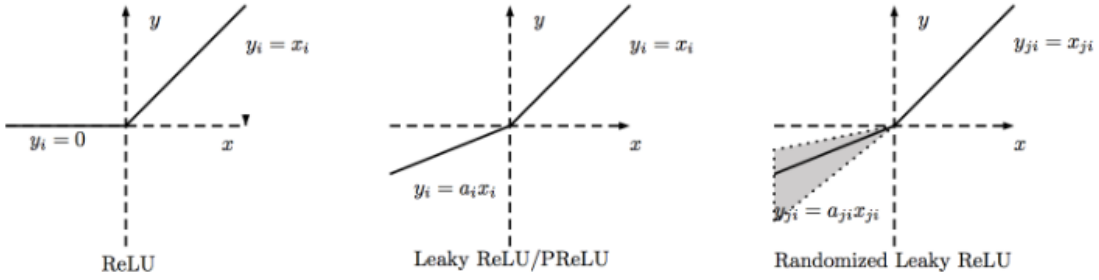


Fig. 5.5. En la izquierda, la ReLU estándar. En el centro, Leaky/paramétrica ReLU con su α definida. A la derecha, la ReLU randomizada, la zona gris define el rango de valores aleatorios que puede tomar α en el proceso de entrenamiento.

1.2.2.6. Softmax

La función Softmax, se trata de una función aplicada en la última capa de una red neuronal, para realizar la clasificación de distintos elementos. La entrada de esta función es un vector con *j* elementos con una cierta 'puntuación'. El resultado de esta función es una distribución de probabilidades (distribución categórica), dentro de un vector con todos los elementos *K* especificados **(1.6)**. Por lo tanto, el resultado tiene que estar definido entre [0, 1].

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=0}^{K} e^{z_k}} \quad \forall \text{ j ... K} \qquad \textbf{(1.6)}$$

1.2.2.7. Otras funciones de activación

En los apartados anteriores se han detallado las funciones de activación más típicas a la hora de realizar una red neuronal. Sin embargo, existe un largo listado con una gran variedad de funciones para aplicaciones más específicas, aunque para la realización de este proyecto no se han utilizado.

1.3. Arquitectura de una red neuronal convolucional

La principal diferencia entre una red neuronal estándar y una convolucional es que en la red convolucional ya desde un principio se especifica que la entrada será una imagen, o en su defecto, una matriz de *MxN* elementos. Las capas de este tipo de redes están situadas en 3 dimensiones (ancho, alto, profundidad). Esto reduce enormemente el número de parámetros de una red. Por esta razón se ha elegido realizar este tipo de red (CNN).

1.3.1. Distribución de las capas

Cualquier red neuronal convolucional se estructura en 4 etapas bien diferenciadas **(1.6)**. En primer lugar, tendríamos la capa de entrada, donde se recoge la dimensión de la imagen (M, N, X) y mantendrá los valores de los píxeles inalterados. A continuación, se procesará la imagen por la llamada, capa convolucional, donde cada neurona se especializará en una región local de la entrada. En este punto se aplicarán ciertos filtros, explicados adelante con más precisión, donde es procesamiento dará como resultado un mapa de características que, superpuestos, dará un conjunto de salidas para ser procesado posteriormente. En tercer lugar, habrá una capa oculta, llamada *Hidden Layer*, donde hay un conjunto de neuronas totalmente conectadas. En estas capas será donde se especificarán los pesos de las neuronas durante el proceso de entrenamiento. Finalmente, la capa de salida, que dará como resultado las puntuaciones de cada clase y realizará el proceso de clasificación.

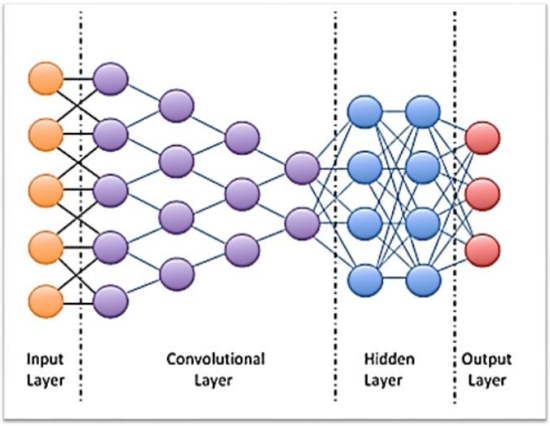


Fig. 6.6. Estructura básica de una CNN.

1.3.2. Descripción de las distintas capas y procesamiento de la imagen

En este apartado se especificará con más precisión cada capa aplicada en el proyecto.

1.3.2.1. Capa convolucional

Como se ha hablado antes, esta capa es la que diferencia una red convolucional a cualquier otra red neuronal. Es la encargada realizar el mapa de características mediante filtros y la que conlleva una carga computacional mayor. Es decir, son filtros que pueden aprender. Cada filtro se aplicará para una región en concreto de la imagen y la recorrerá, a lo largo y ancho, para realizar este mapa. Los filtros se activan cada vez que detectan una característica visual, como bordes, orientaciones, manchas…

Existen tres hiperparámetros que controlan la capa convolucional:

- La profundidad de la salida (*depth*), corresponde a la cantidad de filtros que se quieren aplicar. Es el conjunto de neuronas que están mirando la misma región de la entrada.

- El paso del deslice de filtro (*stride*), que indica cuantos pixeles a la vez se va a mover el filtro para recorrer la imagen.

- Relleno de ceros (*zero-padding*), permite rellenar de ceros el alrededor del borde y de esta manera controlar el tamaño del volumen de salida.

En Keras, la declaración de la capa convolucional (Conv2D) es de la siguiente manera:

```
keras.layers.Conv2D(filters, kernel_size, activation='')
```

- **filters:** Cantidad de filtros (*depth*).
- **kernel_size:** Tamaño de la ventana para realizar la convolución.
- **activation:** Función de activación deseada.

1.3.2.2. Capa Pooling

La capa Pooling típicamente se inserta entre una agrupación de capas convolucionales. Su función es reducir el tamaño espacial de la representación, es decir, equivalente a reducir el cálculo dentro de la red utilizando la operación MAX. Por ejemplo, si utilizamos una capa Maxpooling 2x2, lo que realmente estamos realizando es una reducción de los datos a la mitad, obteniendo el valor máximo dentro de un rango de píxeles.

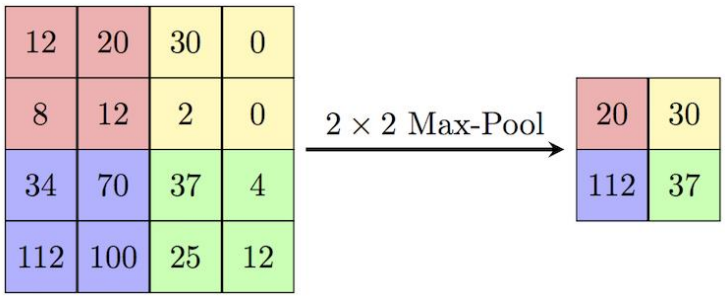


Fig. 7.7. Ejemplo gráfico de operar con una 2x2 Max-Pool.

En Keras, la declaración de la capa pooling (MaxPooling2D) es de la siguiente manera:

```
keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None,
                          data_format='')
```

- **pool_size:** Número que define la reducción, en caso de especificar un tipo (2, 2), se reduce a la mitad el largo y ancho de la imagen.

- **strides:** Tamaño de la ventana para realizar la operación.

- **data_format:** Formato de los datos de la imagen, puede venir primero especificado el tamaño y luego su volumen o viceversa.

1.3.2.3. Capa Zero-Padding

La capa Zero-Padding es utilizada para el relleno de ceros alrededor del borde. Sirve para conservar la altura y la anchura de las imágenes durante el procesado y de esta manera no preocuparse por el tamaño del tensor. Permite también diseñar redes más profundas ya que sin el relleno, la reducción del tamaño sería más rápida. Además, mejora el rendimiento de la red al mantener la información en las fronteras de la imagen.

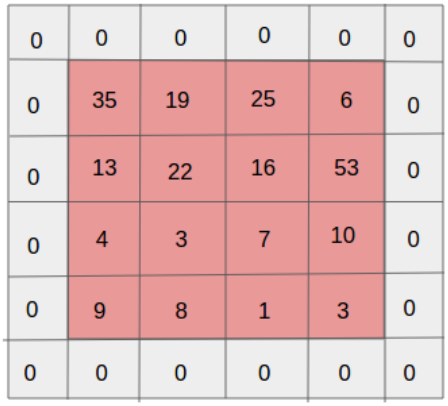


Fig. 8.8. Ejemplo gráfico de añadir Zero-padding a una matriz 4x4.

En Keras, la declaración de la capa zero-padding (ZeroPadding2D) es de la siguiente manera:

```
keras.layers.ZeroPadding2D(padding=(1, 1))
```

- **padding:** Cantidad de filas y columnas a añadir.

1.3.2.4. Capa Flatten

La capa Flatten es la capa intermedia entre las convolucionales y las finales clasificatorias. Permite pasar la información de una matriz a un array para ser procesado posteriormente por las capas que están totalmente interconectadas.

En Keras, la declaración de la capa Flatten es de la siguiente manera:

```
model.add(Flatten())
```

1.3.2.5. Capa Dense/Fully-connected

Esta capa situada previamente ya a la salida, se trata del conjunto de capas que están interconectadas totalmente (**figura 1.9**). Es decir, cada entrada está conectada a una salida con su respectivo valor en el peso.

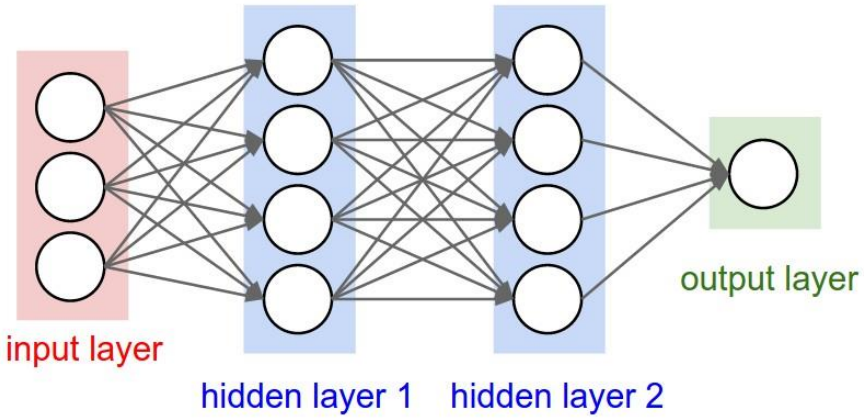


Fig. 9.9. Ejemplo de un conjunto de capas totalmente conectadas (hidden layer 1 con la hidden layer 2).

En Keras, la declaración de la capa dense es de la siguiente manera:

```
keras.layers.Dense(units, activation='')
```

- **units:** Dimensión deseada a la salida.
- **activation:** Función de activación deseada.

1.4. Arquitecturas propuestas para el proyecto

Para este proyecto se han definido 2 tipos de arquitecturas distintas:

- **VGGnet:** Es un tipo de red neuronal convolucional, propuesta por Karen Simonyan and Andrew Zisserman en su paper *Very Deep Convolutional Networks for Large-Scale Image Recognition*[2]. Algo que demuestran estas redes neuronales son que la profundidad de una red neuronal es un parámetro crítico. Pero una gran desventaja de estas redes es que tienen muchos parámetros en el momento de entrenamiento.

- **Resnet:** Red residual, fue propuesta por Kaiming He en *Deep Residual Learning for Image Recognition*[3]. Se caracterizan por el uso intensivo del

[2] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. Retrieved from http://arxiv.org/abs/1409.1556

[3] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. Retrieved from http://arxiv.org/abs/1512.03385

batch normalization, que es una técnica para mejorar el rendimiento y la estabilidad de las redes neuronales artificiales.

A continuación, se especificará la arquitectura programada.

1.4.2. VGG-16 y VGG-19

Dentro de las VGGnet se ha decidido implementar la VGG-16 (**10.10)** y VGG-19 (**11.11)**.

- VGG-16 está compuesta por un total de 16 capas, 13 de ellas de convolución, 2 de ellas totalmente conectadas y la capa final una softmax para clasificar.

- VGG-19 está compuesta por un total de 19 capas, 16 de ellas de convolución, 2 de ellas totalmente conectadas y la capa final una softmax para clasificar.

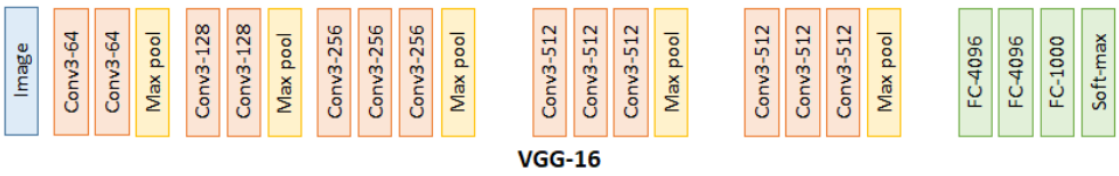


Fig. 12.10. Arquitectura en capas de una red VGG-16. Hace falta especificar que para este proyecto la última capa tiene 7 elementos no 1000.

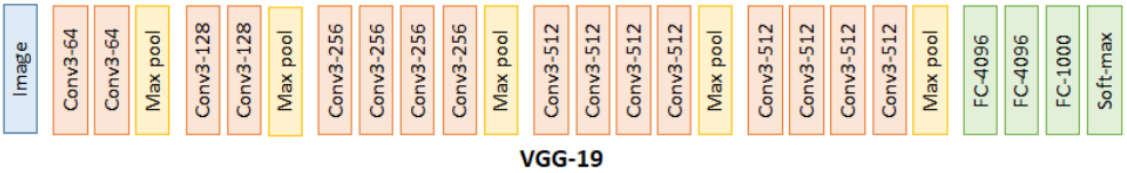


Fig. 13.11. Arquitectura en capas de una red VGG-19. Hace falta especificar que para este proyecto la última capa tiene 7 elementos no 1000.

1.4.3. Resnet50

Resnet50 está compuesta por un total de 50 capas compuestas en bloques e interconectadas. Su arquitectura **(1.12)** por bloques convolucionales que se repiten, en total hay 5 tipos de estos bloques y sumando todas las capas son 50 en total.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 14.11. Arquitectura en capas de la arquitectura Resnet, concretamente utilizaremos la 50-layer siendo la Resnet50.

1.5. Conclusión

Llegados a este punto, donde tenemos una visión global sobre que es una neurona, las funciones que le podemos aplicar a la hora del entrenamiento y su estructura en capas, es hora de ver que algoritmos y parámetros tenemos disponibles para que nuestra red neuronal, entrene de la manera más eficiente posible. En el próximo capítulo se habla justamente sobre esto, los algoritmos de optimización desarrollados y que podemos implementar en nuestro proyecto.

Capítulo 2. Algoritmos de optimización

2.1. Introducción

El objetivo de entrenar una red neuronal es obtener los pesos asociados *(W).* Los algoritmos de optimización nos permiten maximizar el rendimiento de un modelo al ajustar iterativamente sus parámetros hasta minimizar la función objetivo. Este ajuste de los parámetros permite minimizar la pérdida en el proceso de aprendizaje.

Por esta razón existen diferentes estrategias para la optimización y distintos tipos de algoritmos para cada modelo. A continuación, se exponen los principales algoritmos.

2.2. Algoritmos de optimización de descenso de gradiente

2.2.1. Descenso de gradiente

El descenso de gradiente es un algoritmo de optimización de primer orden utilizado para encontrar el mínimo en una función. El descenso de gradiente es usado para actualizar los pesos *(W)* en un modelo de red neuronal para poder minimizar la función de pérdida. Para encontrar el mínimo de una función que utiliza el descenso de gradiente, se dan pasos proporcionales al negativo del gradiente de la función $-\nabla F(a)$. Si consideramos el error que tenemos en función de los posibles pesos, podemos graficar la función como un paraboloide elíptico, tal y como se muestra en la **figura 2.1.**

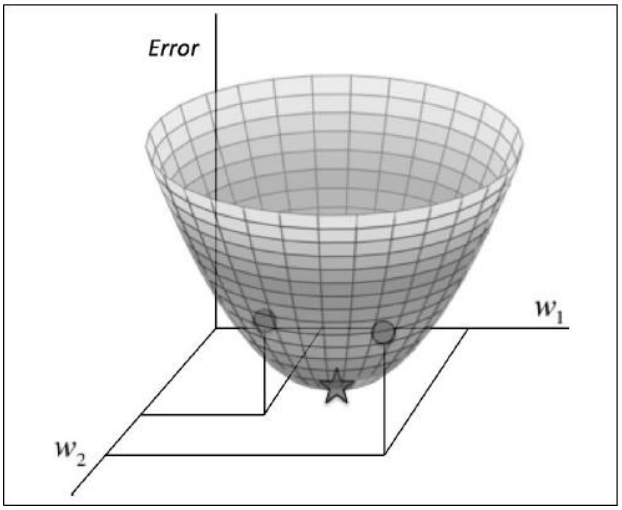


Fig. 2.1. Superficie del error cuadrático para una neurona

En este punto la estrategia a seguir seria inicializar los pesos aleatoriamente para partir de un punto inicial. Seguidamente evaluar el gradiente en la posición

actual y encontrar la dirección de descenso más inclinado. Realizando una reevaluación de la dirección con más pendiente descendiente en la nueva posición el objetivo sería finalizar en la posición mínima de la función.

2.2.2. Tasa de aprendizaje

Para asignar los pesos a nuestro modelo, los algoritmos de optimización requieren de los llamados híper-parámetros para el proceso de entrenamiento. Uno de ellos es la tasa de aprendizaje (η).

Este parámetro de aprendizaje (η) le dice al algoritmo de optimización como mover los pesos *(W)* en la dirección del gradiente para una cantidad de datos *(Batch)*. Elegir este parámetro correctamente es crucial para optimizar correctamente nuestro modelo, ya que si la tasa de aprendizaje es demasiado alta puede no converger hacía el mínimo de la función, hasta incluso divergir, y si es demasiado pequeña el descenso del gradiente puede ser demasiado lento.

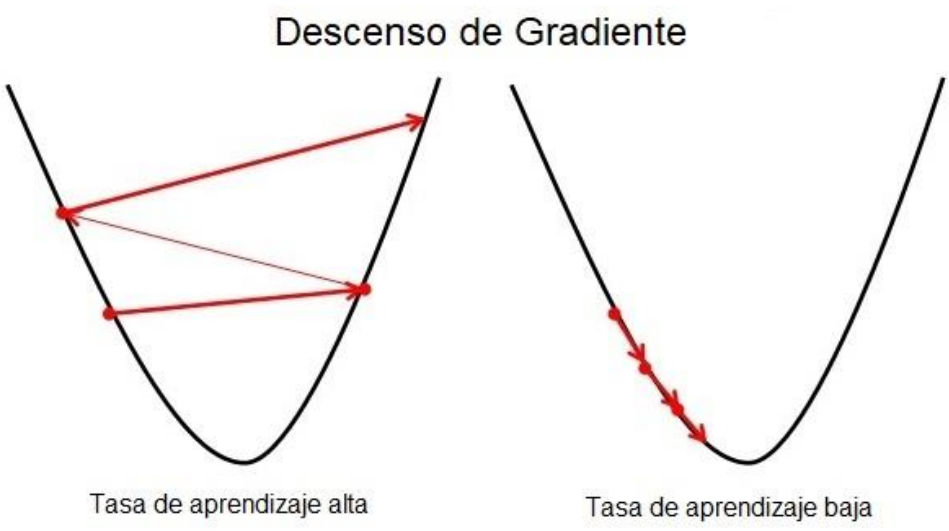


Fig. 2.2. Representación gráfica de los pasos dados por las diferentes tasas de aprendizaje.

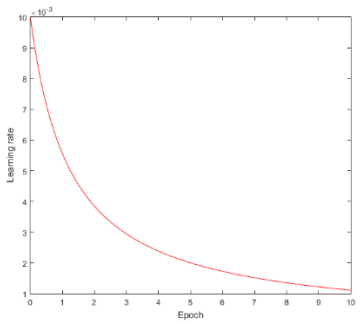
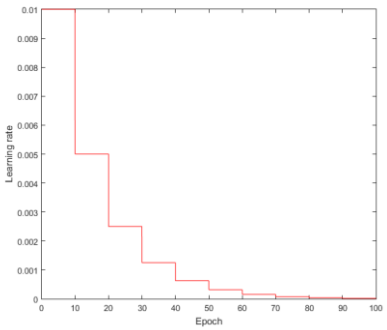
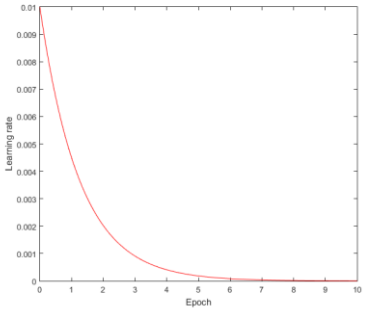
Su relación con el gradiente ∇ y el punto de la función (a_n), se representa con la siguiente expresión matemática.

$$a_{n+1} = a_n - \eta \nabla \mathrm{F}(a_n) \qquad \textbf{(2.1)}$$

Se puede establecer una tasa de aprendizaje inicial (η_0). Los valores típicos para una red neuronal van desde más pequeño que 1 y más grande que 10^{-6}. Aunque establecer un valor en concreto no es lo más óptimo.

Una manera más inteligente es definir una estrategia o programa para obtener una tasa de aprendizaje adaptativa. A continuación, se adjunta una tabla con las distintas maneras de obtener una tasa de aprendizaje adaptativa.

Taula 2.1. Tabla donde se exponen distintas tasas de aprendizaje dinámicas.

Nombre	Fórmula	Gráfico
Decaimiento basado en tiempo	$\eta_{n+1} = \frac{\eta_n}{1+kt}$ **(2.2)**	
Decaimiento basado en pasos	$\eta_{n+1} = \eta_n \, d^{\left\lfloor \frac{epoch}{epoch\_d} \right\rfloor}$ **(2.3)**	
Decaimiento exponencial	$\eta_{n+1} = \eta_n e^{-kt}$ **(2.4)**	

En el decaimiento basado en tiempo **(2.2)**, *k* es un híper-parámetro asociado al algoritmo de optimización y *t* es el número de iteración. En el decaimiento basado en pasos **(2.3)**, *d* es el parámetro *drop* que especifica cuanto va a caer la tasa de aprendizaje, *epoch* es el número de ciclos de entrenamiento y *epoch_d* cada cuantos ciclos de entrenamiento va a realizar la caída. En el decaimiento exponencial **(2.4)**, *k* nuevamente es un híper-parámetro y *t* es el número de iteración.

2.2.3. Momentum

El *momentum* (γ) es un parámetro que permite acelerar el descenso de gradiente estocástico (SGD).

$$a_{n+1} = \gamma a_n - \eta \nabla F(a_n) \quad \textbf{(2.5)}$$

Cuando se usa el momento, básicamente se está acelerando la velocidad de empuje hacia el mínimo, cada vez volviéndose más rápido. Como resultado, obtenemos una convergencia más rápida y reducimos la oscilación durante el aprendizaje.

2.2.4. Gradiente acelerado Nesterov

El gradiente acelerado Nesterov (NAG) es un parámetro que nos permite controlar el momento de una manera más inteligente. Controla la velocidad de descenso en función de la pendiente, por ejemplo, en el caso de que haya una subida, que sepa ralentizar el momento.

De esta manera el gradiente acelerado Nesterov le da al momento esa capacidad de predicción. Sabiendo el momento, si añadimos a la fórmula **(2.5)** esta mejora seremos capaces de tener una aproximación para la siguiente posición.

$$a_{n+1} = \gamma a_n - \eta \nabla F(a_n - \gamma a_n) \quad \textbf{(2.6)}$$

Esta mejora en la anticipación previene de ir demasiado rápido y nos proporciona una mayor capacidad de respuesta, lo que ofrece un mejor rendimiento de las redes neuronales.

2.2.5. Descenso de gradiente estocástico

El descenso de gradiente estocástico (SGD), también conocido como descenso de gradiente incremental, es un algoritmo de optimización que realiza una actualización de los parámetros para cada ejemplo de entrenamiento $x^{(i)}$ y su etiqueta $y^{(i)}$.

$$a_{n+1} = a_n - \eta \nabla F(a_n; x^{(i)}; y^{(i)}) \quad \textbf{(2.7)}$$

A diferencia del descenso de gradiente, que converge al mínimo de la función para la configuración de los parámetros, el descenso de gradiente estocástico puede saltar a otros mínimos locales potencialmente mejores.

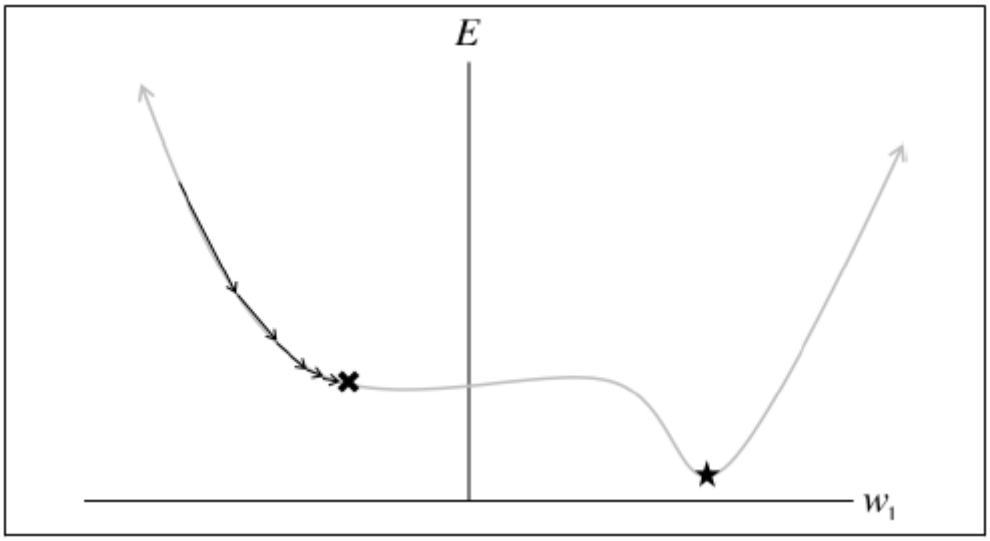


Fig. 2.3. El descenso de gradiente convencional es sensible a los mínimos locales, en cambio un uso del SGD nos podría permitir saltar al mínimo global (marcado con una estrella) en este caso.

En Keras, la declaración del descenso de gradiente estocástico es de la siguiente manera:

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

- **lr:** Es la tasa de aprendizaje (η), valores mayores o igual a 0.
- **momentum:** Momento (γ), valores mayores o igual a 0.
- **decay:** Decaimiento de la tasa de aprendizaje.
- **nesterov:** Gradiente acelerado Nesterov (NAG).

2.2.6. Adagrad

Adagrad es un algoritmo de optimización, propuesto por primera vez por Duchi en 2011[4], que adapta la tasa de aprendizaje global durante todo el tiempo de entrenamiento usando la acumulación de los anteriores gradientes. Realiza actualizaciones más grandes para datos infrecuentes y actualizaciones más pequeñas par datos frecuentes.

Esta tasa de aprendizaje se calcula respecto a la media cuadrática (RMS) de los gradientes anteriores.

[4] Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12.Jul (2011): 2121-2159.

El Adagrad se puede expresar matemáticamente de la siguiente forma.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ij}+\epsilon}} \nabla_{\theta_t} \mathrm{F}(\theta_{t,i}) \quad \textbf{(2.8)}$$

- **$\nabla_\Theta F(\Theta_{t,i})$:** Gradiente de la función objetivo.
- **$G_{t,ij}$:** Matriz diagonal que contiene la media cuadrática de los gradientes.
- **ϵ:** Termino que permite evitar la división por 0, toma un valor de $1 \cdot 10^{-8}$.

En Keras, la declaración del Adagrad es de la siguiente manera:

```
keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
```

- **lr:** Es la tasa de aprendizaje (η), valores mayores o igual a 0.
- **epsilon:** Epsilon (ϵ), valores mayores o igual a 0.
- **decay:** Decaimiento de la tasa de aprendizaje.

2.2.7. Adadelta

Adadelta es una extensión del Adagrad que trata de reducir su tasa de aprendizaje de una manera menos agresiva. En lugar de acumular todos los gradientes pasados, se establece una ventana que acumula los últimos gradientes.

En Keras, la declaración del Adadelta es de la siguiente manera:

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)
```

- **lr:** Es la tasa de aprendizaje (η), valores mayores o igual a 0.
- **rho:** Es la constante de decaimiento similar a la usada en el *momentum*.
- **epsilon:** Epsilon (ϵ), valores mayores o igual a 0.
- **decay:** Decaimiento de la tasa de aprendizaje.

2.2.8. RMSProp

RMSProp es un algoritmo de optimización que trata de resolver el problema de la disminución radical del Adagrad. La conclusión es que la solución de usar la acumulación de gradientes no es suficiente.

RMSProp propone recuperar el concepto de amortiguar las fluctuaciones en el gradiente. Si en vez realizar una acumulación de gradientes esta vez realizamos la media móvil exponencial (EWMA), nos permite de manera automática descartar los valores que se hicieron hace mucho tiempo.

En Keras, la declaración del RMSProp es de la siguiente manera:

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

- **lr:** Es la tasa de aprendizaje (η), valores mayores o igual a 0.
- **rho:** Es la constante de decaimiento similar a la usada en el *momentum*.
- **epsilon:** Epsilon (ϵ), valores mayores o igual a 0.
- **decay:** Decaimiento de la tasa de aprendizaje.

2.2.9. Adam

Adam es un algoritmo de optimización resultado de la combinación del RMSProp y *momentum*.

En Keras, la declaración del RMSProp es de la siguiente manera:

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
        epsilon=None, decay=0.0, amsgrad=False)
```

- **lr:** Es la tasa de aprendizaje (η), valores mayores o igual a 0.
- **beta_1:**
- **beta_2:**
- **epsilon:** Epsilon (ϵ), valores mayores o igual a 0.
- **decay:** Decaimiento de la tasa de aprendizaje.
- **amsgrad:** Aplicar la variante AMSGrad [5]del algoritmo.

2.3. Conclusión

En este capítulo hemos realizado un repaso en todos los algoritmos de optimización. No se ha explorado la opción de los algoritmos de optimización de segundo grado, debido a su complejidad y dificultad de aplicación. En el siguiente capítulo, describiremos brevemente la base de datos que ha sido utilizada para el entrenamiento de nuestra red neuronal.

[5] Sashank J. Reddi, Satyen Kale & Sanjiv Kuma ″On the Convergence of Adam and Beyond″ *Published as a conference paper at International Conference on Learning Representations (ICLR),* 2018.

Capítulo 3. Desarrollo de la red neuronal

3.1. Base de datos

Los datos consisten en imágenes en escala de grises de 48x48 píxeles de caras. Las caras se han registrado automáticamente para que la cara esté más o menos centrada y ocupe aproximadamente la misma cantidad de espacio en cada imagen. La tarea es categorizar cada rostro según la emoción mostrada en la expresión.

La base de datos está estructurada en un .csv y contiene 2 columnas, la emoción y los pixeles. La columna "emoción" contiene un código numérico que va de 0 a 6, para la emoción que está presente en la imagen. La columna "píxeles" contiene una cadena rodeada de comillas para cada imagen. Los contenidos de esta cadena son valores de píxeles separados por espacios en el orden mayor de la fila. test.csv contiene solo la columna "píxeles" y su tarea es predecir la columna de emoción.

El conjunto de entrenamiento consta de 28.709 ejemplos. El conjunto de pruebas públicas utilizado para la tabla de clasificación consta de 3.589 ejemplos. El conjunto de prueba final, que se usó para determinar el ganador de la competencia, consta de otros 3,589 ejemplos.

Código	Número de imágenes	Emoción
0	4593	Enfadado
1	547	Asco
2	5121	Miedo
3	8989	Feliz
4	6077	Triste
5	4002	Sorpresa
6	6198	Neutral

Tabla 3.1. Tabla donde se muestra el número de imágenes según cada emoción.

Fig. 3.1. Ejemplo de las distintas emociones dentro de la base de datos fer2013.

3.2. Configuración y entrenamiento

Se ha desarrollado un script para el entrenamiento de los modelos. Además, dentro de este script se han definido unas variables para la configuración del entrenamiento. A continuación, se muestran las capturas de pantalla del programa.

```
26  # ----------------------------MODEL CONFIGURATION--------------------------------#
27  weightsDirectory = '../weights/vgg19_weights1.h5'
28
29  # SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam
30  optimizer = 'Adam'
31  lr = 0.0001
32  decay = 0.0
33
34  epsilon = None  # RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam
35  rho = 0.95  # RMSprop, Adadelta
36  momentum = 0.0  # SGD
37  nesterov = True  # SGD
38  amsgrad = False  # Adam
39  # ------------------------------------------------------------------------------#
```

Fig. 3.2. Configuración del modelo.

Dentro de la configuración del modelo, **(3.2)** se puede seleccionar, la ruta de los pesos, que anteriormente, si se ha realizado algún entrenamiento, se puede cargar para un reentrenamiento. En las siguientes líneas, tenemos la posibilidad de la selección del tipo de algoritmo de optimización que queremos, la tasa de aprendizaje, el decaimiento (aunque más adelante se realiza una función dónde se puede especificar la fórmula deseada para una tasa de aprendizaje dinámica) y para finalizar los hiperparámetros característicos de cada algoritmo.

```
53  # ------------------------------PREPROCESS DATA----------------------------------#
54  x = np.load('../database/facial_data_X.npy')  # Pixels
55  y = np.load('../database/facial_labels.npy')  # Emotions
56
57  # Normalizing pixels values
58  x -= np.mean(x, axis=0)
59  x /= np.std(x, axis=0)
60
61  # Converts a class vector (integers) to binary class matrix.
62  y_ = np_utils.to_categorical(y, num_classes=7)
63
64  X_train = x[0:28710, :]
65  X_crossval = x[28710:32300, :]
66
67  X_train = X_train.reshape((X_train.shape[0], 1, 48, 48))
68  X_crossval = X_crossval.reshape((X_crossval.shape[0], 1, 48, 48))
69
70  Y_train = y_[:28710]  # 28710
71  Y_crossval = y_[28710:32300]
72  # ------------------------------------------------------------------------------#
```

Fig. 3.3. Pre procesamiento de las imágenes

Dentro del pre procesamiento de las imágenes **(3.3)** se cargan los valores de los píxeles y sus respectivas etiquetas, en un formato de vector *numpy*. A continuación, se normalizan sus valores, para un cálculo más rápido. Finalmente se separan los datos que pertenecen estrictamente para el entrenamiento y los que pertenecen al proceso de test posterior.

Una vez se ha realizado la configuración necesaria, estamos listos para realizar el entrenamiento de nuestra red. Aquí hay que tener en cuenta que se obtendrán varios resultados, como la precisión durante el entrenamiento, la función de pérdidas… Pero a nosotros nos interesa la precisión del proceso de test final, que nos indicara con que precisión nuestra red neuronal "acierta". Realizado varios entrenamientos con distintas configuraciones, se ha obtenido el siguiente resultado con las diferentes arquitecturas de redes neuronales **(Tabla 3.2)**.

	VGG-16	**VGG-19**	**Resnet50**
Tasa de aprendizaje	$0.1x10^{-3}$	$1x10^{-3}$	$1x10^{-3}$
Optimizador	Adam	Adam	Adam
Decaimiento	Temporal	Pasos	Pasos
Ciclos de entrenamiento	12	10	10
Precisión de Test	**60.02%**	55.26%	54.34%

Tabla 3.2. Tabla de los mejores resultados de precisión de test con los parámetros utilizados.

Como se puede apreciar el mejor resultado que hemos obtenido es en la arquitectura VGG-16. A partir de ahora trabajaremos con esta arquitectura para el resto de resultados.

3.3. Testing de la red neuronal

Una vez tenemos la red neuronal entrenada, es el momento de realizar pruebas con ella. En primer lugar, se ha desarrollado un programa que permite detectar caras en una fotografía y extraerlas en pequeñas imágenes de 48x48 píxeles. I a continuación se introduce esta imagen por nuestra red neuronal para mostrar los resultados. Los resultados están representados en un diagrama de barras con las probabilidades de cada emoción. La emoción con mayor probabilidad es por la que se decide la red. A continuación, se muestra el proceso de una imagen propia para verificar el funcionamiento de la red ()

Fig. 3.4. Imagen original.

Fig. 3.5. Imagen con cara detectada.

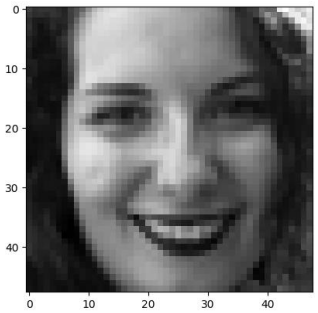


Fig. 3.6. Imagen de la cara con la que trabajará nuestra red neuronal.

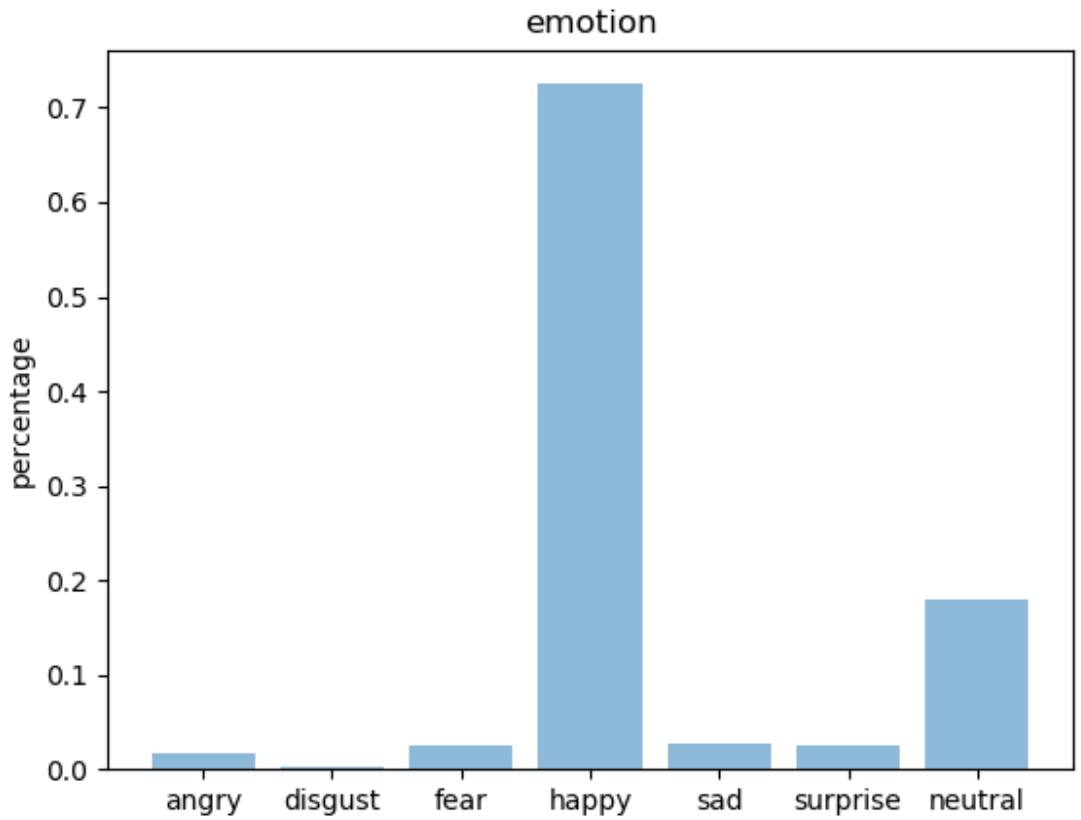


Fig. 3.7. Gráfica de barras que muestra las probabilidades de las emociones que ha extraído nuestra res neuronal.

Como podemos observar, el resultado nos muestra la emoción 'felicidad' alrededor del 71%, que realmente es el resultado esperado. Después la segunda emoción que determina más probable es la neutralidad con un 20%. El resto de emociones las podemos considerar despreciables.

Una vez realizado varios test y observar que nuestra red neuronal sabe identificar de manera correcta la mayoría de imágenes, estamos preparados para obtener los resultados con la base de datos de políticos proporcionada.

Capítulo 4. Resultados

4.1. Realización de los resultados

Para la extracción de resultados se ha decidido realizar el análisis de las emociones presentadas en 6 políticos diferentes. Las imágenes han sido extraídas de las principales cadenas de televisión del país durante los telediarios del mes de septiembre del año 2017.

4.2. Emociones presentadas

Para poder realizar un análisis de múltiples imágenes, se ha modificado el script para que, en vez de mostrar el diagrama de barras, tenga como salida la emoción con más probabilidad. Entonces todos estos resultados, números entre el 0 y el 6 (0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral), se han extraído en un Excel para mostrarlos en una gráfica **(4.1)** y tabla **(4.1)**.

PUIGDEMONT 09/2017		
	FREQ ABS.	FREQ REL.
Angry	308	2,65%
Disgust	0	0,00%
fear	2489	21,43%
happy	531	4,57%
sad	2242	19,30%
surprise	25	0,22%
neutral	6022	51,84%
TOTAL	11617	

ALBERT RIVERA 09/2017		
	FREQ ABS.	FREQ REL.
Angry	357	18,43%
Disgust	0	0,00%
fear	265	13,68%
happy	111	5,73%
sad	190	9,81%
surprise	0	0,00%
neutral	1014	52,35%
TOTAL	1937	

RAJOY 09/2017		
	FREQ ABS.	FREQ REL.
Angry	1918	11,53%
Disgust	0	0,00%
fear	1504	9,04%
happy	2074	12,46%
sad	2619	15,74%
surprise	4	0,02%
neutral	8521	51,21%
TOTAL	16640	

JUNQUERAS 09/2017		
	FREQ ABS.	FREQ REL.
Angry	140	2,96%
Disgust	0	0,00%
fear	1294	27,39%
happy	142	3,01%
sad	550	11,64%
surprise	5	0,11%
neutral	2593	54,89%
TOTAL	4724	

ARRIMADAS 09/2017		
	FREQ ABS.	FREQ REL.
Angry	61	3,18%
Disgust	0	0,00%
fear	50	2,60%
happy	286	14,89%
sad	114	5,93%
surprise	3	0,16%
neutral	1407	73,24%
TOTAL	1921	

ICETA 09/2017		
	FREQ ABS.	FREQ REL.
Angry	202	9,01%
Disgust	0	0,00%
fear	297	13,24%
happy	623	27,78%
sad	196	8,74%
surprise	11	0,49%
neutral	914	40,75%
TOTAL	2243	

Tabla. 4.1. Resultados presentados por los distintos políticos.

En la tabla se muestra el total de imágenes usadas para el análisis y las frecuencias absolutas y relativas de cada emoción. Los siguientes ejemplos **(4.2)** han sido etiquetados por la red neuronal.

Fig. 4.2. Distintos ejemplos utilizados para el análisis. Arriba a la izquierda Puigdemont neutral, arriba en el centro Albert Rivera feliz, arriba a la derecha Rajoy enfadado, abajo a la izquierda Junqueras triste, abajo centro Arrimadas feliz y abajo a la derecha Iceta con miedo.

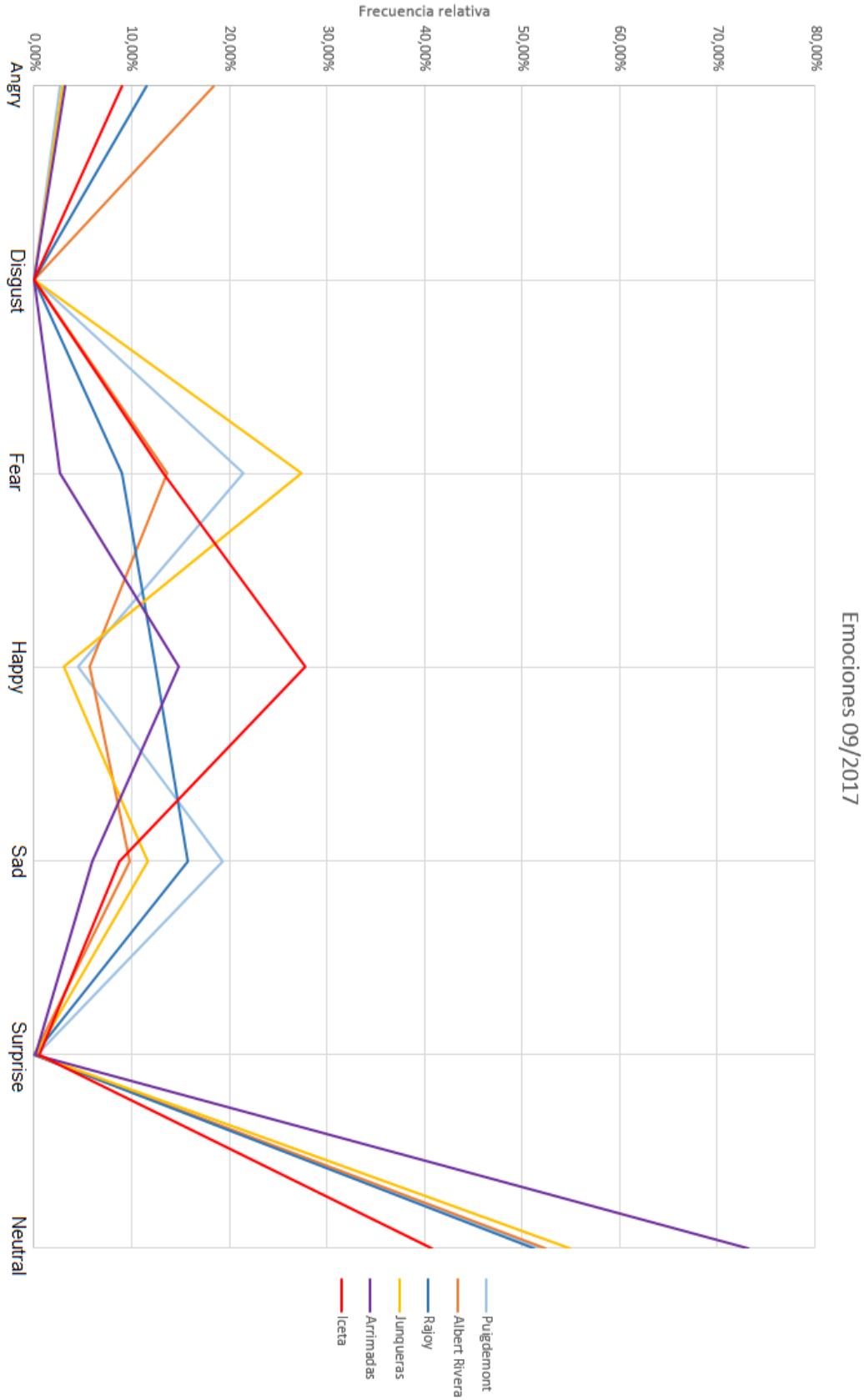


Fig.4.1 Gráfica que muestra los resultados obtenidos. Cada color representa un político, en el eje vertical encontramos la frecuencia relativa.

Conclusiones

- En primer lugar, se ha logrado construir un sistema capaz de reconocer rostros humanos y catalogar 7 emociones expresadas con una precisión del alrededor del 60%. Podemos afirmar que hemos conseguido desarrollar una red neuronal convolucional para obtener la expresión facial. Sin embargo, el sistema es mejorable, factores como la base de datos para el entrenamiento o la mejor optimización de los parámetros podría mejorar el sistema.

- Se han explorado las distintas arquitecturas propuestas, obteniendo distintos resultados, siendo el más satisfactorio la red neuronal VGG-16.

- Se han explorado las distintas funciones de activación, capas, técnicas y algoritmos de optimización para la mayor optimización de nuestra red, obteniendo una mejora de los resultados a lo largo del tiempo.

- A pesar del overfiting de nuestra red, se ha logrado reducir su efecto mediante técnicas de dropout y pre procesamiento de la base de datos.

- Se ha logrado extender una aplicación real, extrayendo más datos de una base de datos real como la de *http://politics.ugiat.com/.*

- Una vez llegados aquí podemos concluir que los conocimientos teóricos han sido adquiridos para poder llegar a realizar este proyecto y ser capaces de desarrollar una red neuronal convolucional.

Bibliografía

[1] Al-Shabi, M., Cheah, W. P., & Connie, T. (2016). Facial Expression Recognition Using a Hybrid CNN-SIFT Aggregator. Recuperado de http://arxiv.org/abs/1608.02833

[2] Burkert, P., Trier, F., Afzal, M. Z., Dengel, A., & Liwicki, M. (2015). DeXpression: Deep Convolutional Neural Network for Expression Recognition.

[3] Ding, H., Zhou, S. K., & Chellappa, R. (2016). FaceNet2ExpNet: Regularizing a Deep Face Recognition Net for Expression Recognition. Recuperado de http://arxiv.org/abs/1609.06591

[4] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. *MIT Press*. Recuperado de http://www.deeplearningbook.org

[5] He, K., Zhang, X., Ren, S., & Sun, J. (2015a). Deep Residual Learning for Image Recognition. Recuperado de http://arxiv.org/abs/1512.03385

[6] He, K., Zhang, X., Ren, S., & Sun, J. (2015b). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Recuperado de http://arxiv.org/abs/1502.01852

[7] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Recuperado de http://arxiv.org/abs/1502.03167

[8] Janocha, K., & Czarnecki, W. M. (2017). On Loss Functions for Deep Neural Networks in Classification. Recuperado de https://arxiv.org/abs/1702.05659

[9] Ruder, S. (2016). An overview of gradient descent optimization algorithms. Recuperado de https://arxiv.org/abs/1609.04747

[10] Savoiu, A., & Wong, J. (2017). Recognizing Facial Expressions Using Deep Learning.

[11] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. Recuperado de http://arxiv.org/abs/1409.1556

[12] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting.